



A-Z80 CPU User's Guide

An FPGA project recreating the Z80

© 2018 Goran Devic

3/10/2018

Revision History

Date	Revision	Change
2014-12-14	1.0	Initial revision.
2014-12-21	1.1	Added ZX Spectrum Turbo mode, speaker blink etc.
2015-01-22	1.2	Added section on file generators; other small updates.
2016-03-06	1.3	Added Xilinx support.
2017-01-07	1.4	Corrections and update.
2018-03-10	1.5	Corrections and update.

Table of Contents

Revision History	1
Introduction	3
Project Directory Structure	4
Environment	5
Test Boards	6
Simulation	7
Module simulations	7
Top-level simulations	10
Verification.....	12
Fuse tests	12
Selected functional tests.....	13
Z80 Assembly level tests	14
Tools.....	15
PLA Checker Tool	15
Arduino Tools.....	18
Integration	19
Interface.....	20
Sample Implementations	21
Simple host.....	21
Sinclair ZX Spectrum	24
Advanced Topics	26
Modifying the A-Z80 CPU Core	26
File Generators.....	28
Building the CPU	28
Verification/Tests.....	30

Introduction

A-Z80 is a conceptual implementation of the venerable Zilog® Z80 processor targeted to synthesize and run on a modern FPGA device. It differs from the existing Z80 implementations in that it is designed from the ground-up through the schematics and low-level gates.

This design is capable of mimicking the actual Z80 CPU and, through its architecture, it illustrates its inner workings.

The A-Z80 implementation strives to be internally structurally identical to the original Z80. Using this approach, the model achieves a full cycle accuracy and has identical behavior for all documented and undocumented features (*). This is achieved not by explicitly hard-coding exceptions and quirks, but by mimicking the actual design.

Various *Zilog Z80* references are widely available so the CPU, its instructions and behavior will not be covered in this document.

This document focuses on the structure and mechanics of working with the A-Z80 architecture; it should help you understand it and incorporate it into your designs.

You can read more about the conception and implementation of the A-Z80 on its home website: www.baltazarstudios.com .

If you simply want to integrate A-Z80 into your own project, you can skip to section *Integration*.

Project Directory Structure

A-Z80 project can be downloaded from two locations:

- OPENCORES (SVN): <http://opencores.org/project,a-z80>
- Bitbucket (git): <https://bitbucket.org/gdevic/a-z80> .

The following table describes the directory structure:

Directory	Sub-directory	Description
cpu		Contains all core files of the A-Z80 CPU
	alu	Arithmetical and Logical Unit files
	bus	Various bus-related files
	control	Control unit files
	registers	Register block files
	toplevel	A-Z80 top level interfaces and projects
docs		Documentation and images of schematics
host		Several implementations using A-Z80 on Altera and Xilinx boards
	basic_de1	Basic computer for testing and verification, uses UART, Altera DE1
	basic_nexys3	The same but for Xilinx Nexys3 board
	zxspectrum_de1	Sinclair ZX Spectrum implementation for Altera DE1 board
resources		General project resources and scripts
tools		Building and testing utilities and misc. files
	Arduino	Software for Arduino Mega dongle to interface with a Z80
	dongle	Dongle and simulation scripts and golden files
	z80_pla_checker	Windows utility to test and create A-Z80 PLA tables
	zmac	Z80 test and verification assembler files

Environment

A minimal set of tools needed to **compile** various parts of the project is:

- Altera Quartus II Web Edition (free) OR
- Xilinx ISE Webpack (free) OR
- Lattice ICECube toolchain from Synopsys
- ModelSim (Altera edition, free) – needed for simulation of various modules
- Python 3.5 (or newer) – needed to change or recompile A-Z80 files
- Microsoft Visual Studio 2013 or newer – needed to recompile the `z80_pla_checker` tool

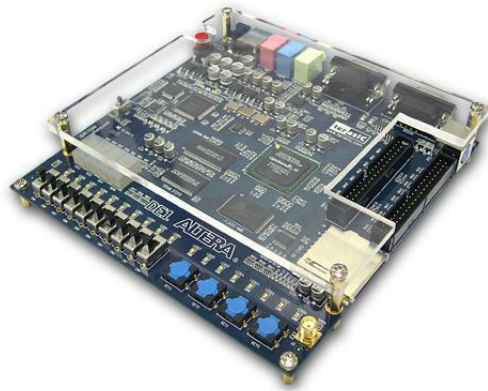
If you are only going **to use** A-Z80 in your project, all Verilog files that you need are already checked in and ready to be included and you don't need any of the extra tools listed. Use Python script `"export.py"` to copy/export necessary files for you (don't copy files manually since only some files are needed and the script 'knows' which ones).

This project was developed and tested on a 64-bit Windows 7 OS. Your mileage may vary on other platforms.

Test Boards

Altera: Designs are tested on a Terasic DE1 board:

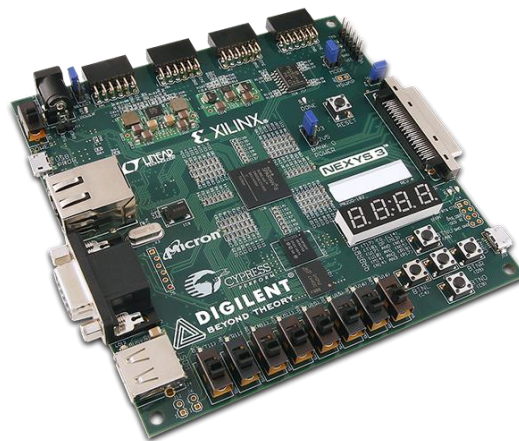
<http://www.altera.com/education/univ/materials/boards/de1/unv-de1-board.html>



This board has a **Cyclone II EP2C20F484C7** FPGA alongside several useful peripherals including a 512 KB SRAM bank, PS/2 keyboard, UART and a VGA connector.

Xilinx: Designs are tested on a Digilent Nexys3 board:

<http://store.digilentinc.com/nexys-3-spartan-6-fpga-trainer-board-limited-time-see-nexys4-ddr/>



This board features a **Spartan-6 XC6LX16-CS324** FPGA and several useful peripherals such as UART, PS/2 keyboard and VGA.

Simulation

Module simulations

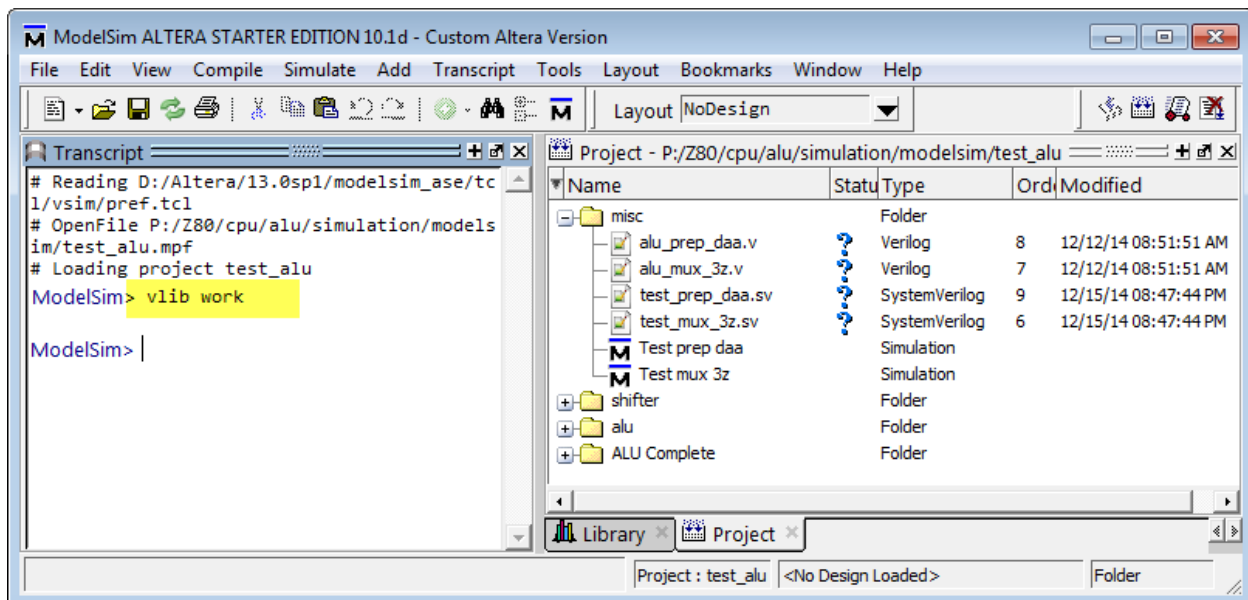
If you make any change to the CPU core files, you should run one or more simulations to verify the correctness of your modifications.

Each module in the “**cpu**” directory contains a ModelSim simulation project that verifies the functionality of one or more of its blocks. Before opening a project in ModelSim, run “**modelsim_setup.py**” script located in the project root directory. That script will set up *relative file mappings* to enable project to reside anywhere on your drive.

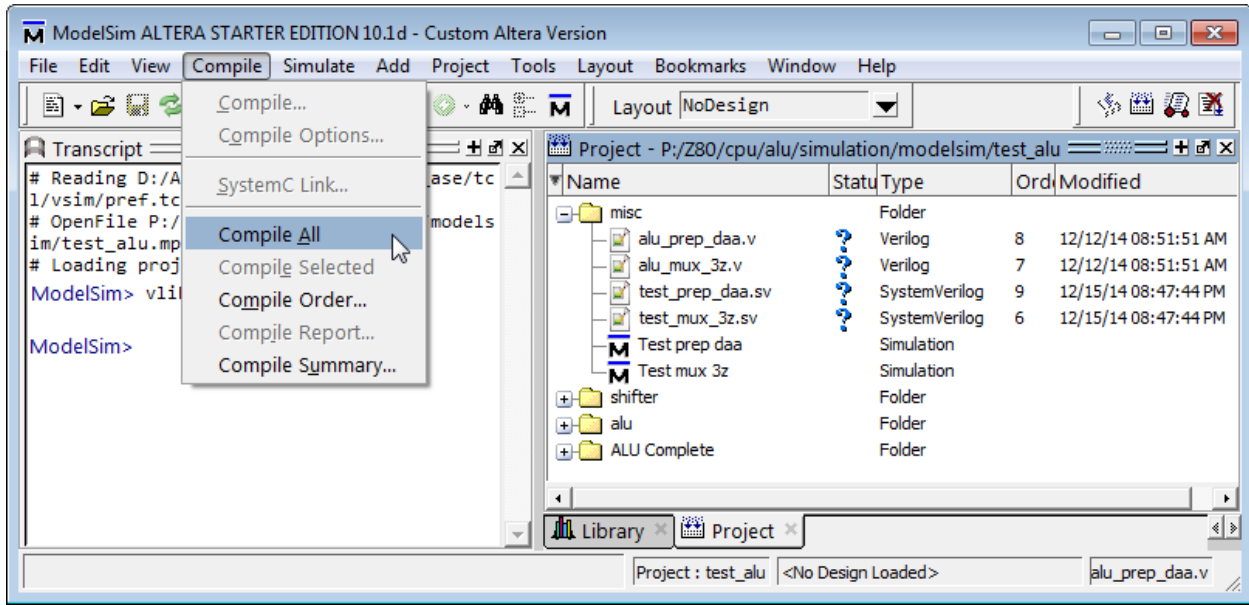
If you have installed and configured ModelSim correctly, double-clicking on any ***.mpf** file will open a project in the ModelSim GUI.

This example will illustrate setting up and starting a simulation of a specific logic block in the **alu** module.

Before you can compile a simulation test bench, create a library by typing “**vlib work**” as shown:



Next, select “**Compile->Compile All**” to compile all files that are part of a module simulation.



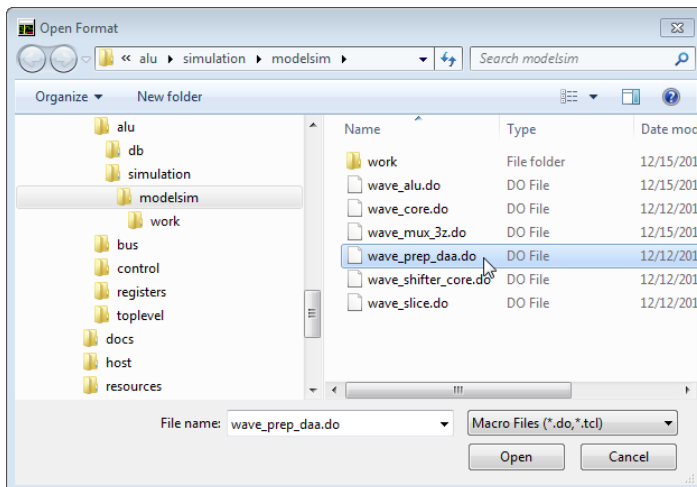
Each project has one or more *simulation configurations*; each configuration tests a specific block of logic. In addition, each configuration has its own wave file which you can load before you run a simulation. Wave files are customized for a specific test and provide a handy way to quickly see all relevant signals.

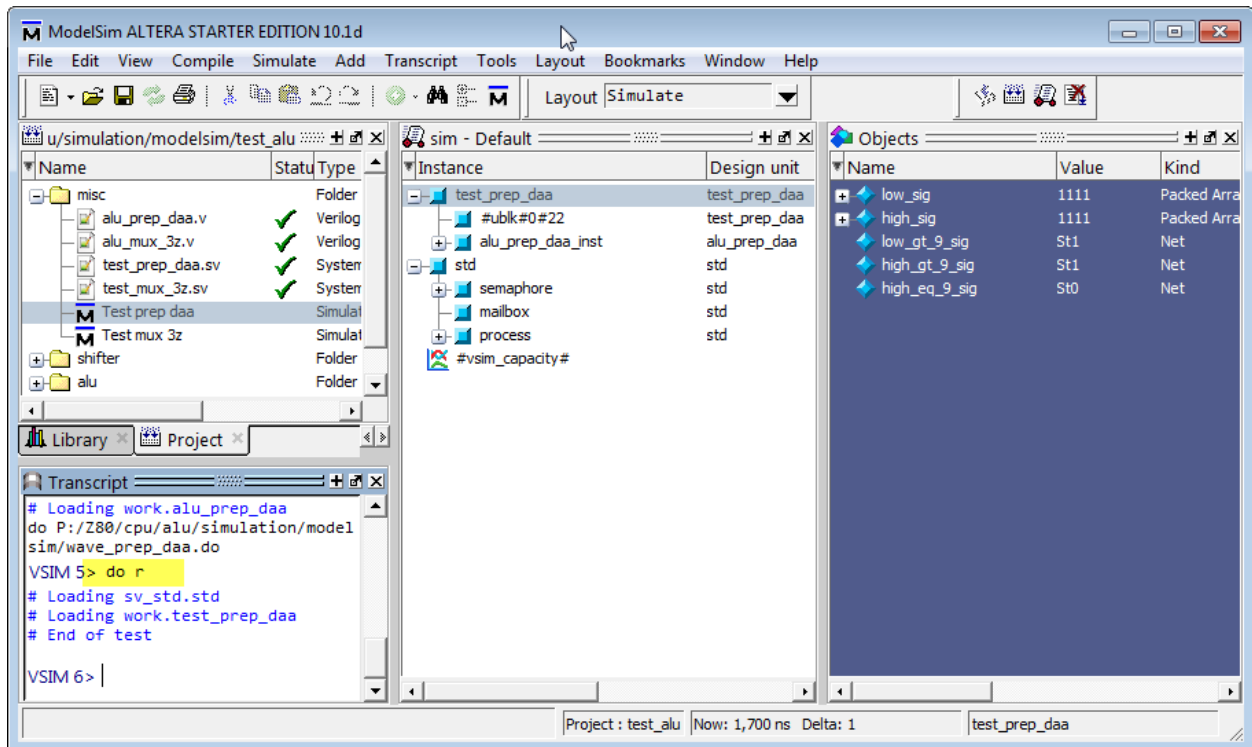
In this example, we will run “Test prep daa” configuration. DAA is a Z80 instruction that adjusts accumulator for a decimal operation. It requires calculating the adjustment addend based on the result of a previous operation. Hence, this test is written to verify the correctness of that calculation.

Each test configuration is run by a main test bench file that is always written in a *System Verilog* language with the extension *.sv. A file that runs the “Test prep daa” configuration is “**test_prep_daa.sv**”.

Double-click on the “Test prep daa” configuration and your simulation should be loaded.

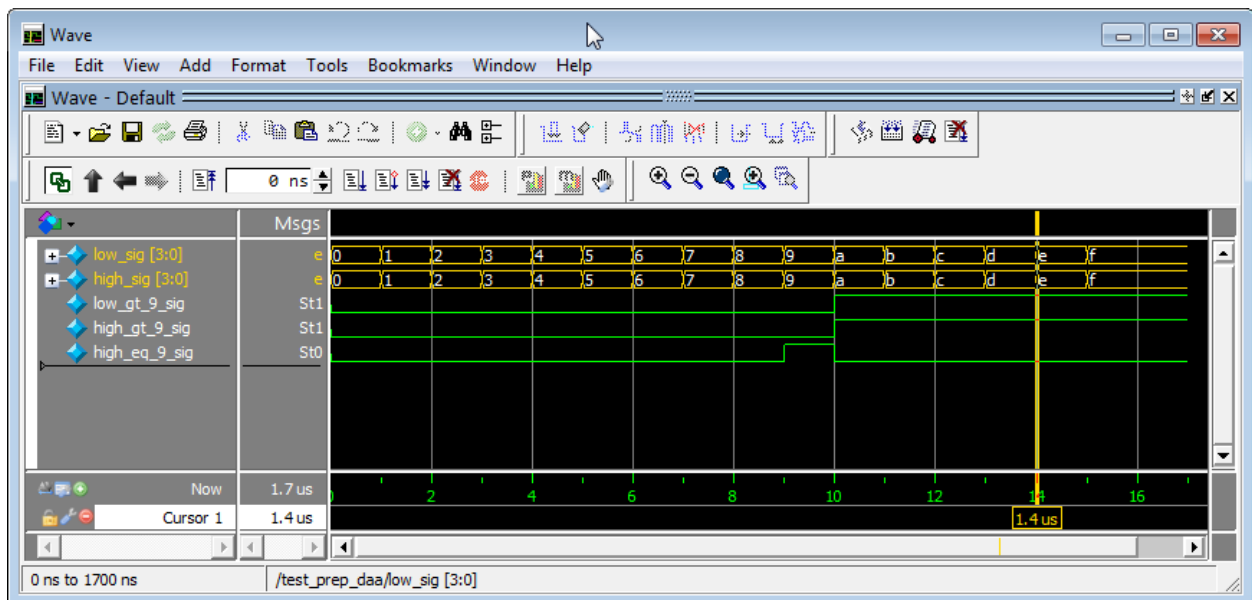
Open the wave window if it is not already visible and select File->Open to load a wave file as shown:





A shortcut is provided to run simulation: each ModelSim directory contains a small text file with the name “r” that contains command “**restart -f ; run -all**”. Run, or rerun, a test simply by typing “**do r**” as shown above.

After running this example, you should see a waveform of the DAA preparation block:



Although a very simple, this example hopefully illustrated a method of running a simulation. The same process can be repeated with other configurations and modules. The pattern of configurations, files and waveform names is the same.

Each main test bench file (just like the “**test_prep_daa.sv**”) contains a set of **assert()** statements to verify the signal correctness. These asserts will fail and your simulation will stop if the signals take unexpected values.

Most simulations run for a predetermined number of clocks. The exceptions are top-level simulations (in the directory “**cpu\toplevel\simulation\modelsim**”) and a basic host simulation (in the directory “**host\basic_de1\simulation\modelsim**”). These simulations need to be stopped manually since they simply continue to execute given Z80 executable code.

Top-level simulations

The two top-level simulations are designed to load an arbitrary Z80 assembly code and execute it. A simple unidirectional UART model is provided for the Z80 software to write to the ModelSim console. The UART model will simulate behavior of a synthesized serial port. When the same design is synthesized for the FPGA, the same Z80 code will write messages through a physical serial port.

UART is using 115200 baud data transfer rate, 8 data, 1 stop bit, no parity.

Module	Simulation project
Toplevel	cpu\toplevel\simulation\modelsim\test_top.mpf
Basic host	host\basic_de1\simulation\modelsim\test_host.mpf

Those two simulation configurations can run any Z80 code; several sample test sources can be found in the directory “**tools\zmac**” along with ZMAC assembler and a few batch scripts that simplify compilation and the test setup. Z80 test files are somewhat based on CP/M and have a BDOS style text print interface.

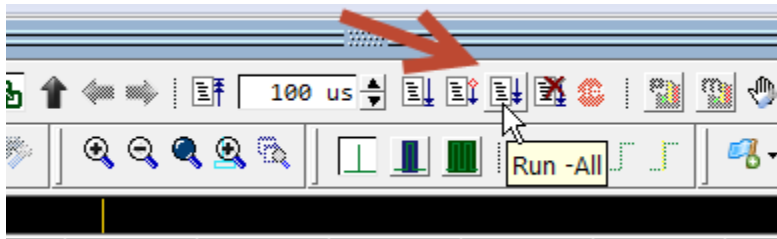
Two MS DOS batch files are used to compile and run tests:

Batch file	Description
tools\zmac\make_modelsim.bat	Compiles and generates executable code for a ModelSim test at “cpu\toplevel\simulation\modelsim\test_top.mpf”, for “test_top” configuration.
tools\zmac\make_fpga.bat	<ol style="list-style-type: none"> 1. Compiles and generates executable code in Intel HEX file format to be included into the target FPGA data file for basic host “host\basic_de1\ host_board.qpf” 2. Also generates executable code for the basic host ModelSim test at “host\basic_de1\simulation\modelsim\ test_host.mpf”

You can create your own Z80 assembly tests to run: simply drag and drop an assembly source file onto one of those two batch files and they will compile a file and copy the results into proper target directories.

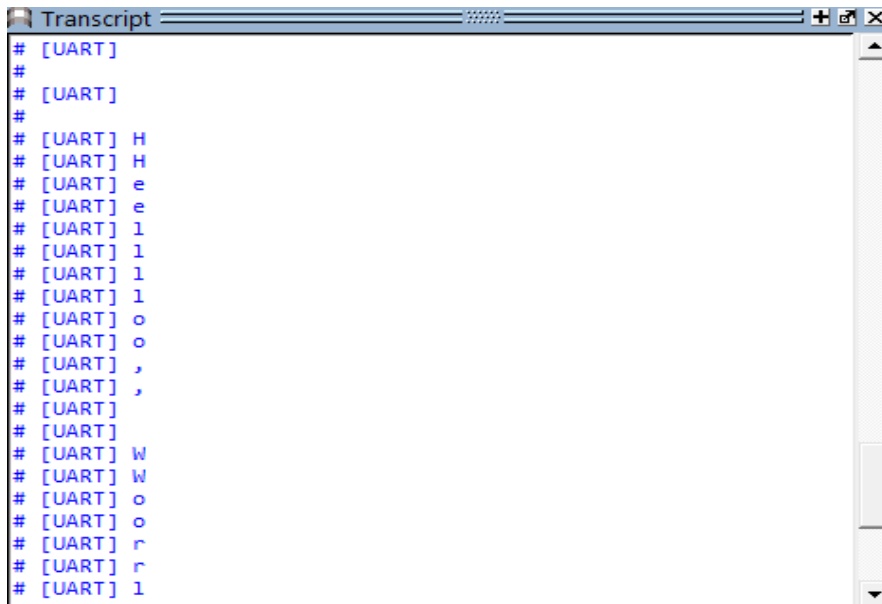
For this example, we will compile and run a “Hello, world” test (“`tools\zmac\hello_world.asm`”).

Drag and drop “`hello_world.asm`” onto the “`make_modelsim.bat`” and start a top-level simulation (“`test_top`” configuration) in the ModelSim.



Shortly, you should see the output in the ModelSim console window.

After you see the text being written to the virtual UART device, you can stop the simulation.

A screenshot of the ModelSim Transcript window. The window title is 'Transcript'. The output text is as follows:

```
# [UART]
# [UART]
# [UART] H
# [UART] H
# [UART] e
# [UART] e
# [UART] l
# [UART] l
# [UART] l
# [UART] l
# [UART] o
# [UART] o
# [UART] ,
# [UART] ,
# [UART]
# [UART]
# [UART] W
# [UART] W
# [UART] o
# [UART] o
# [UART] r
# [UART] r
# [UART] l
```

Verification

Fuse tests

Fuse is a set of tests to verify Z80 at the individual instruction level. Written for software emulator designers, it contains a fairly complete set of input and output states for each instruction.

Files that are used in this verification are subset of the Fuse emulator source package: <http://fuse-emulator.sourceforge.net>. You can find them in the “`cpu\toplevel\fuse`” directory.

The files describe individual instruction's tests and need to be processed into a format that we can run – which is Verilog. A Python script “`cpu\toplevel\genfuse.py`” generates Verilog test code for a selected number of Fuse tests.

See that script file for more details on how to configure it before running.

When run, it creates “`cpu\toplevel\test_fuse.vh`” include file.

```
// Automatically generated by genfuse.py
force dut.reg_file_.reg_gp_we=0;
force dut.reg_control_.ctl_reg_sys_we=0;
force dut.z80_top_ifc_n.fpga_reset=1;
#2 //-----
    force dut.instruction_reg_.ctl_ir_we=1;
    force dut.instruction_reg_.db=0;
#2 release dut.instruction_reg_.ctl_ir_we;
    release dut.instruction_reg_.db;
$fdisplay(f,"Testing opcode 00      NOP");
...
```

Once generated, this include file needs to be compiled with a ModelSim project file “`cpu\toplevel\simulation\modelsim\test_top.mpf`” to run a set of tests. The test output will show in the ModelSim window and the test will also create and write a file “`fuse.result.txt`”.

Hint: You can speed up Fuse simulation if you disable output to the wave window by typing:

```
VSIM 10> nolog -all
```

The following command re-enables the output:

```
VSIM 10> nolog -reset
```

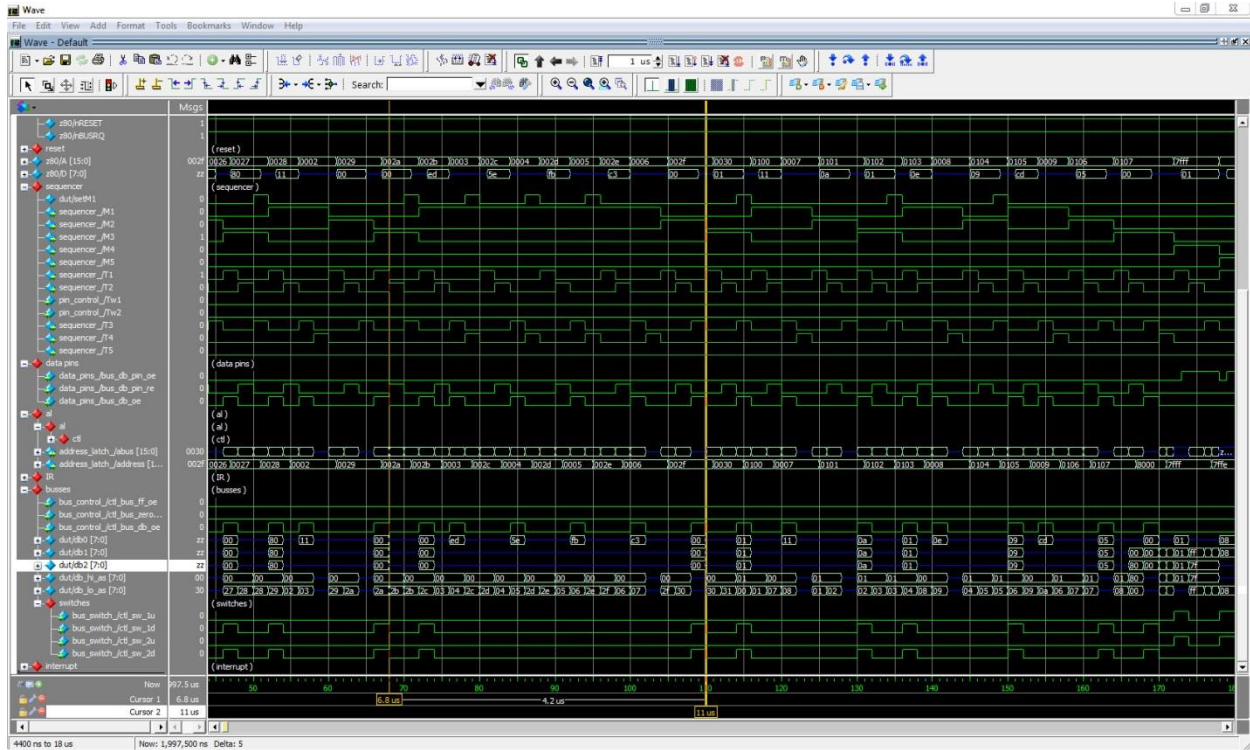


Image 1 : Fuse tests in ModelSim

Results of Fuse tests are written in the file “**fuse.result.txt**”, one instruction per line:

```

Testing opcode 00      NOP
Testing opcode ed67   RRD
Testing opcode ed6f   RLD
Testing opcode 81     ADD A,C
Testing opcode cb41   BIT 0,C
Testing opcode cb93   RES 2,E
...
    
```

Selected functional tests

There are 3 tests that verify specific ALU operations by cross-checking the results run on a real Z80 with the algorithm written in Python:

Test directory	Z80 test file	Description
tools\dongle\daa	tools\zmac\test.daa.asm	Execute DAA instruction for all values 0-255
tools\dongle\neg	tools\zmac\test.neg.asm	Execute NEG instruction for all values 0-255
tools\dongle\sbc	n/a	Simulate SUB and SBC instructions

Python scripts run the Arduino Z80 dongle (described in the Tools section) and generate output files. Those files are then compared with the output produced by another set of Python scripts (they implement corresponding algorithms). Lastly, the same text files are compared with ModelSim

simulation of those instructions and by running the same executable on the Simple Host FPGA implementation and capturing the UART output.

The “golden” files include values of flags and accumulator going into the instruction and the result after the instruction has completed:

```
F:00 A:00 -> 00 F:44
F:00 A:01 -> 01 F:00
F:00 A:02 -> 02 F:00
F:00 A:03 -> 03 F:04
F:00 A:04 -> 04 F:00
F:00 A:05 -> 05 F:04
F:00 A:06 -> 06 F:04
F:00 A:07 -> 07 F:00
...
```

Z80 Assembly level tests

Folder “**tools/zmac**” contains several Z80 assembly level tests.

Test source file	Description
tools\zmac\hello_world.asm	A mandatory “Hello, World”
tools\zmac\zexdoc.asm	Tests documented Z80 instructions and flags
tools\zmac\zexall.asm	Tests ALL Z80 instructions and flags (documented and undocumented)

While all of them can run in ModelSim, the last two are comprehensive and very long tests which should normally be run only on an FPGA hardware in full speed mode.

“**hello_world.asm**” is written to allow test bench “**cpu\toplevel\test_top.sv**” to exercise various interrupt modes. It contains interrupt handlers and logging for the test bench to run the following cases:

- Inject a single or periodic NMI
- Inject a single or periodic INT
- Test response to the nWAIT signal
- Test response to the nBUSRQ signal
- Test resets

Tools

PLA Checker Tool

PLA checker tool in “**tools\z80_pla_checker**” directory is a test utility to verify and create PLA code used to statically decode Z80 instruction groups.

Since you normally don't have to compile that code, the executable file is checked in. This is a .NET application which should equally well run on Windows and Linux with mono support.

The PLA checker tool loads several files from the “**resources**” directory. That includes a raw PLA table definition which was reverse-engineered from an image of a Z80 die.

```

Z80 PLA
File
Modifiers: IX0 IX1 NHALT ALU XX CB ED | cls redo
PLA Checker Tool Copyright (C) 2014 Goran Devic
This program comes with ABSOLUTELY NO WARRANTY
This is free software and you are welcome to redistribute it under certain conditions;
-----
Loading PLA: ../../resources/z80-pla.txt
Total 105 PLA lines
Loading opcode table: ../../resources\opcodes-xx.txt
Loading opcode table: ../../resources\opcodes-cb-xx.txt
Loading opcode table: ../../resources\opcodes-ed-xx.txt
Loading opcode table: ../../resources\opcodes-dd-xx.txt
Loading opcode table: ../../resources\opcodes-dd-cb.txt

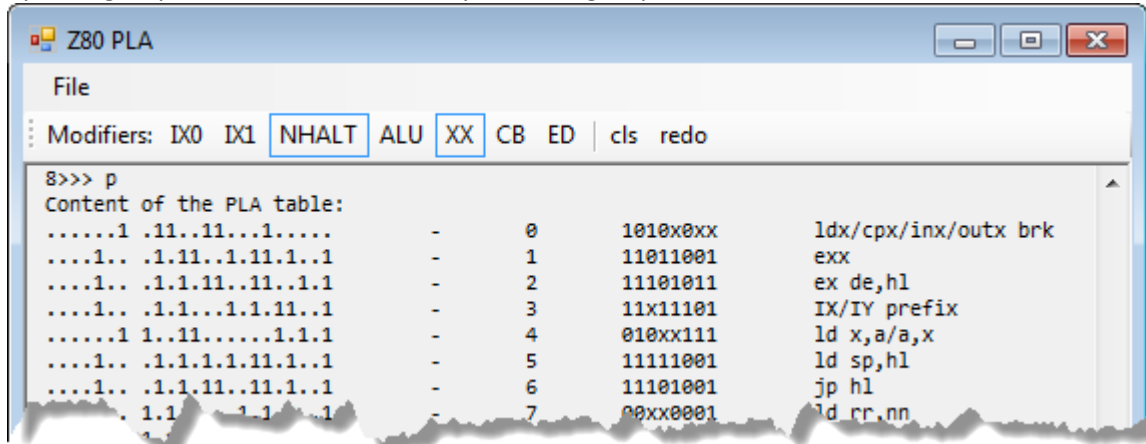
p          - Dump the content of the PLA table
p [#]     - For a given PLA entry # (dec) show opcodes that trigger it
m [#]     - Match opcode # (hex) with a PLA entry (or match 0-FF)
g         - Generate a Verilog PLA module
t [#] <#> - Show opcode table in various ways
          0 - Display number of PLA entries that trigger on each opcode
          1 - For each opcode, display all PLA entry numbers that trigger
          <#> - Add a * to opcodes for which the specified PLA entry triggers
q 101000... Query PLA table string
c         - Clear the screen

```

The tool was invaluable in the development of A-Z80 and maintain its value as a cross-checker for the PLA code. Available commands are:

Cmd	Description
h or ?	Help, list all commands.
p	PLA table contains a set of modifiers and a gate-level logic array that 'filters' various instruction

opcode groups. This command shows you those groups.



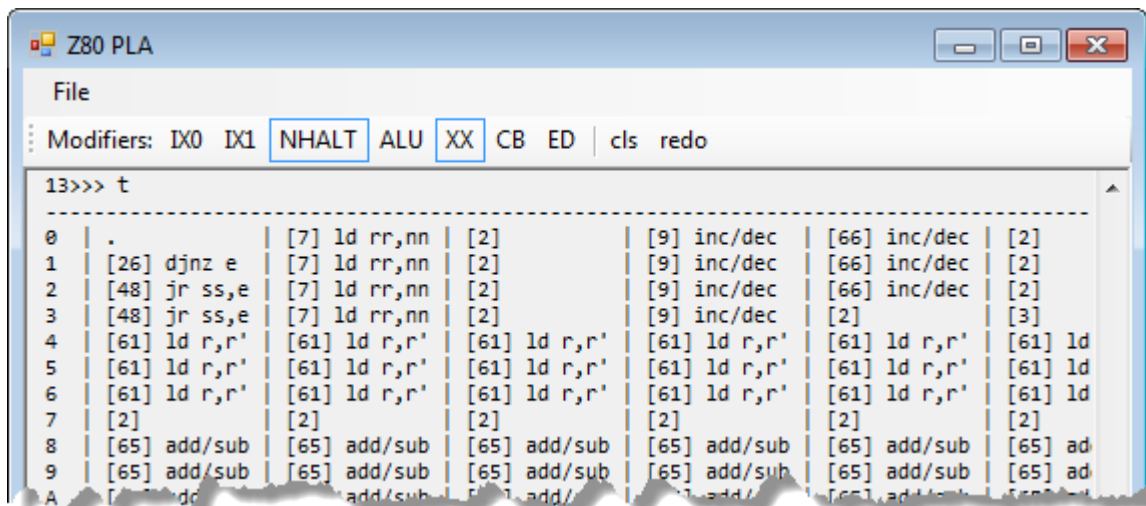
p # Given a PLA entry number (decimal), show opcodes that are activated by it



m # This is a reverse-lookup that shows all PLA table entries that would activate a specific opcode given as a hex number:



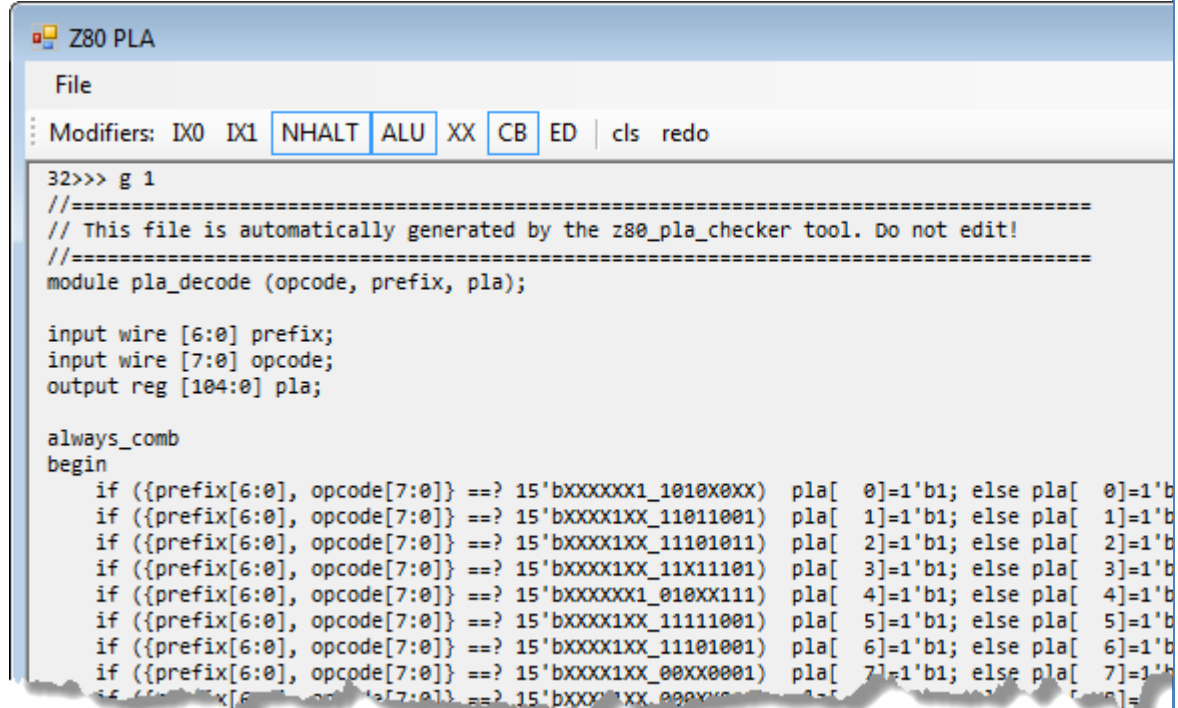
t Dumps the opcode table in several ways. One or two optional arguments are given which restrict the table or show extra information including the number of PLA entries that trigger for each opcode etc.



q # Useful only while simulating the CPU design, this command decodes the actual PLA table string which is a long sequence of binary digits (105 bits in total)

g Generates Verilog code that implements the PLA decode. The output of this command is used to

create “cpu\control\pla_decode.sv” source file which is at the core of the design.



```

Z80 PLA
File
Modifiers: IX0 IX1 NHALT ALU XX CB ED | cls redo
32>>> g 1
//=====
// This file is automatically generated by the z80_pla_checker tool. Do not edit!
//=====
module pla_decode (opcode, prefix, pla);

input wire [6:0] prefix;
input wire [7:0] opcode;
output reg [104:0] pla;

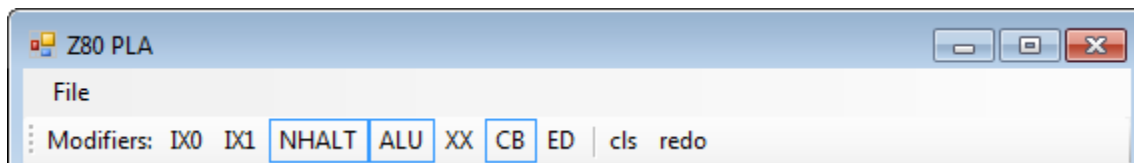
always_comb
begin
  if ({prefix[6:0], opcode[7:0]} ==? 15'bXXXXXX1_1010X0XX) pla[ 0]=1'b1; else pla[ 0]=1'b
  if ({prefix[6:0], opcode[7:0]} ==? 15'bXXXX1XX_11011001) pla[ 1]=1'b1; else pla[ 1]=1'b
  if ({prefix[6:0], opcode[7:0]} ==? 15'bXXXX1XX_11101011) pla[ 2]=1'b1; else pla[ 2]=1'b
  if ({prefix[6:0], opcode[7:0]} ==? 15'bXXXX1XX_11X11101) pla[ 3]=1'b1; else pla[ 3]=1'b
  if ({prefix[6:0], opcode[7:0]} ==? 15'bXXXXXX1_010XX111) pla[ 4]=1'b1; else pla[ 4]=1'b
  if ({prefix[6:0], opcode[7:0]} ==? 15'bXXXX1XX_11111001) pla[ 5]=1'b1; else pla[ 5]=1'b
  if ({prefix[6:0], opcode[7:0]} ==? 15'bXXXX1XX_11101001) pla[ 6]=1'b1; else pla[ 6]=1'b
  if ({prefix[6:0], opcode[7:0]} ==? 15'bXXXX1XX_00XX0001) pla[ 7]=1'b1; else pla[ 7]=1'b
  if ({prefix[6:0], opcode[7:0]} ==? 15'bXXXX1XX_00XX0001) pla[ 8]=1'b1; else pla[ 8]=1'b
  if ({prefix[6:0], opcode[7:0]} ==? 15'bXXXX1XX_00XX0001) pla[ 9]=1'b1; else pla[ 9]=1'b
  if ({prefix[6:0], opcode[7:0]} ==? 15'bXXXX1XX_00XX0001) pla[10]=1'b1; else pla[10]=1'b
  if ({prefix[6:0], opcode[7:0]} ==? 15'bXXXX1XX_00XX0001) pla[11]=1'b1; else pla[11]=1'b
  if ({prefix[6:0], opcode[7:0]} ==? 15'bXXXX1XX_00XX0001) pla[12]=1'b1; else pla[12]=1'b
  if ({prefix[6:0], opcode[7:0]} ==? 15'bXXXX1XX_00XX0001) pla[13]=1'b1; else pla[13]=1'b
  if ({prefix[6:0], opcode[7:0]} ==? 15'bXXXX1XX_00XX0001) pla[14]=1'b1; else pla[14]=1'b
  if ({prefix[6:0], opcode[7:0]} ==? 15'bXXXX1XX_00XX0001) pla[15]=1'b1; else pla[15]=1'b
  if ({prefix[6:0], opcode[7:0]} ==? 15'bXXXX1XX_00XX0001) pla[16]=1'b1; else pla[16]=1'b
  if ({prefix[6:0], opcode[7:0]} ==? 15'bXXXX1XX_00XX0001) pla[17]=1'b1; else pla[17]=1'b
  if ({prefix[6:0], opcode[7:0]} ==? 15'bXXXX1XX_00XX0001) pla[18]=1'b1; else pla[18]=1'b
  if ({prefix[6:0], opcode[7:0]} ==? 15'bXXXX1XX_00XX0001) pla[19]=1'b1; else pla[19]=1'b
  if ({prefix[6:0], opcode[7:0]} ==? 15'bXXXX1XX_00XX0001) pla[20]=1'b1; else pla[20]=1'b
  if ({prefix[6:0], opcode[7:0]} ==? 15'bXXXX1XX_00XX0001) pla[21]=1'b1; else pla[21]=1'b
  if ({prefix[6:0], opcode[7:0]} ==? 15'bXXXX1XX_00XX0001) pla[22]=1'b1; else pla[22]=1'b
  if ({prefix[6:0], opcode[7:0]} ==? 15'bXXXX1XX_00XX0001) pla[23]=1'b1; else pla[23]=1'b
  if ({prefix[6:0], opcode[7:0]} ==? 15'bXXXX1XX_00XX0001) pla[24]=1'b1; else pla[24]=1'b
  if ({prefix[6:0], opcode[7:0]} ==? 15'bXXXX1XX_00XX0001) pla[25]=1'b1; else pla[25]=1'b
  if ({prefix[6:0], opcode[7:0]} ==? 15'bXXXX1XX_00XX0001) pla[26]=1'b1; else pla[26]=1'b
  if ({prefix[6:0], opcode[7:0]} ==? 15'bXXXX1XX_00XX0001) pla[27]=1'b1; else pla[27]=1'b
  if ({prefix[6:0], opcode[7:0]} ==? 15'bXXXX1XX_00XX0001) pla[28]=1'b1; else pla[28]=1'b
  if ({prefix[6:0], opcode[7:0]} ==? 15'bXXXX1XX_00XX0001) pla[29]=1'b1; else pla[29]=1'b
  if ({prefix[6:0], opcode[7:0]} ==? 15'bXXXX1XX_00XX0001) pla[30]=1'b1; else pla[30]=1'b
  if ({prefix[6:0], opcode[7:0]} ==? 15'bXXXX1XX_00XX0001) pla[31]=1'b1; else pla[31]=1'b
  if ({prefix[6:0], opcode[7:0]} ==? 15'bXXXX1XX_00XX0001) pla[32]=1'b1; else pla[32]=1'b
  if ({prefix[6:0], opcode[7:0]} ==? 15'bXXXX1XX_00XX0001) pla[33]=1'b1; else pla[33]=1'b
  if ({prefix[6:0], opcode[7:0]} ==? 15'bXXXX1XX_00XX0001) pla[34]=1'b1; else pla[34]=1'b
  if ({prefix[6:0], opcode[7:0]} ==? 15'bXXXX1XX_00XX0001) pla[35]=1'b1; else pla[35]=1'b
  if ({prefix[6:0], opcode[7:0]} ==? 15'bXXXX1XX_00XX0001) pla[36]=1'b1; else pla[36]=1'b
  if ({prefix[6:0], opcode[7:0]} ==? 15'bXXXX1XX_00XX0001) pla[37]=1'b1; else pla[37]=1'b
  if ({prefix[6:0], opcode[7:0]} ==? 15'bXXXX1XX_00XX0001) pla[38]=1'b1; else pla[38]=1'b
  if ({prefix[6:0], opcode[7:0]} ==? 15'bXXXX1XX_00XX0001) pla[39]=1'b1; else pla[39]=1'b
  if ({prefix[6:0], opcode[7:0]} ==? 15'bXXXX1XX_00XX0001) pla[40]=1'b1; else pla[40]=1'b
  if ({prefix[6:0], opcode[7:0]} ==? 15'bXXXX1XX_00XX0001) pla[41]=1'b1; else pla[41]=1'b
  if ({prefix[6:0], opcode[7:0]} ==? 15'bXXXX1XX_00XX0001) pla[42]=1'b1; else pla[42]=1'b
  if ({prefix[6:0], opcode[7:0]} ==? 15'bXXXX1XX_00XX0001) pla[43]=1'b1; else pla[43]=1'b
  if ({prefix[6:0], opcode[7:0]} ==? 15'bXXXX1XX_00XX0001) pla[44]=1'b1; else pla[44]=1'b
  if ({prefix[6:0], opcode[7:0]} ==? 15'bXXXX1XX_00XX0001) pla[45]=1'b1; else pla[45]=1'b
  if ({prefix[6:0], opcode[7:0]} ==? 15'bXXXX1XX_00XX0001) pla[46]=1'b1; else pla[46]=1'b
  if ({prefix[6:0], opcode[7:0]} ==? 15'bXXXX1XX_00XX0001) pla[47]=1'b1; else pla[47]=1'b
  if ({prefix[6:0], opcode[7:0]} ==? 15'bXXXX1XX_00XX0001) pla[48]=1'b1; else pla[48]=1'b
  if ({prefix[6:0], opcode[7:0]} ==? 15'bXXXX1XX_00XX0001) pla[49]=1'b1; else pla[49]=1'b
  if ({prefix[6:0], opcode[7:0]} ==? 15'bXXXX1XX_00XX0001) pla[50]=1'b1; else pla[50]=1'b
  if ({prefix[6:0], opcode[7:0]} ==? 15'bXXXX1XX_00XX0001) pla[51]=1'b1; else pla[51]=1'b
  if ({prefix[6:0], opcode[7:0]} ==? 15'bXXXX1XX_00XX0001) pla[52]=1'b1; else pla[52]=1'b
  if ({prefix[6:0], opcode[7:0]} ==? 15'bXXXX1XX_00XX0001) pla[53]=1'b1; else pla[53]=1'b
  if ({prefix[6:0], opcode[7:0]} ==? 15'bXXXX1XX_00XX0001) pla[54]=1'b1; else pla[54]=1'b
  if ({prefix[6:0], opcode[7:0]} ==? 15'bXXXX1XX_00XX0001) pla[55]=1'b1; else pla[55]=1'b
  if ({prefix[6:0], opcode[7:0]} ==? 15'bXXXX1XX_00XX0001) pla[56]=1'b1; else pla[56]=1'b
  if ({prefix[6:0], opcode[7:0]} ==? 15'bXXXX1XX_00XX0001) pla[57]=1'b1; else pla[57]=1'b
  if ({prefix[6:0], opcode[7:0]} ==? 15'bXXXX1XX_00XX0001) pla[58]=1'b1; else pla[58]=1'b
  if ({prefix[6:0], opcode[7:0]} ==? 15'bXXXX1XX_00XX0001) pla[59]=1'b1; else pla[59]=1'b
  if ({prefix[6:0], opcode[7:0]} ==? 15'bXXXX1XX_00XX0001) pla[60]=1'b1; else pla[60]=1'b
  if ({prefix[6:0], opcode[7:0]} ==? 15'bXXXX1XX_00XX0001) pla[61]=1'b1; else pla[61]=1'b
  if ({prefix[6:0], opcode[7:0]} ==? 15'bXXXX1XX_00XX0001) pla[62]=1'b1; else pla[62]=1'b
  if ({prefix[6:0], opcode[7:0]} ==? 15'bXXXX1XX_00XX0001) pla[63]=1'b1; else pla[63]=1'b
  if ({prefix[6:0], opcode[7:0]} ==? 15'bXXXX1XX_00XX0001) pla[64]=1'b1; else pla[64]=1'b
  if ({prefix[6:0], opcode[7:0]} ==? 15'bXXXX1XX_00XX0001) pla[65]=1'b1; else pla[65]=1'b
  if ({prefix[6:0], opcode[7:0]} ==? 15'bXXXX1XX_00XX0001) pla[66]=1'b1; else pla[66]=1'b
  if ({prefix[6:0], opcode[7:0]} ==? 15'bXXXX1XX_00XX0001) pla[67]=1'b1; else pla[67]=1'b
  if ({prefix[6:0], opcode[7:0]} ==? 15'bXXXX1XX_00XX0001) pla[68]=1'b1; else pla[68]=1'b
  if ({prefix[6:0], opcode[7:0]} ==? 15'bXXXX1XX_00XX0001) pla[69]=1'b1; else pla[69]=1'b
  if ({prefix[6:0], opcode[7:0]} ==? 15'bXXXX1XX_00XX0001) pla[70]=1'b1; else pla[70]=1'b
  if ({prefix[6:0], opcode[7:0]} ==? 15'bXXXX1XX_00XX0001) pla[71]=1'b1; else pla[71]=1'b
  if ({prefix[6:0], opcode[7:0]} ==? 15'bXXXX1XX_00XX0001) pla[72]=1'b1; else pla[72]=1'b
  if ({prefix[6:0], opcode[7:0]} ==? 15'bXXXX1XX_00XX0001) pla[73]=1'b1; else pla[73]=1'b
  if ({prefix[6:0], opcode[7:0]} ==? 15'bXXXX1XX_00XX0001) pla[74]=1'b1; else pla[74]=1'b
  if ({prefix[6:0], opcode[7:0]} ==? 15'bXXXX1XX_00XX0001) pla[75]=1'b1; else pla[75]=1'b
  if ({prefix[6:0], opcode[7:0]} ==? 15'bXXXX1XX_00XX0001) pla[76]=1'b1; else pla[76]=1'b
  if ({prefix[6:0], opcode[7:0]} ==? 15'bXXXX1XX_00XX0001) pla[77]=1'b1; else pla[77]=1'b
  if ({prefix[6:0], opcode[7:0]} ==? 15'bXXXX1XX_00XX0001) pla[78]=1'b1; else pla[78]=1'b
  if ({prefix[6:0], opcode[7:0]} ==? 15'bXXXX1XX_00XX0001) pla[79]=1'b1; else pla[79]=1'b
  if ({prefix[6:0], opcode[7:0]} ==? 15'bXXXX1XX_00XX0001) pla[80]=1'b1; else pla[80]=1'b
  if ({prefix[6:0], opcode[7:0]} ==? 15'bXXXX1XX_00XX0001) pla[81]=1'b1; else pla[81]=1'b
  if ({prefix[6:0], opcode[7:0]} ==? 15'bXXXX1XX_00XX0001) pla[82]=1'b1; else pla[82]=1'b
  if ({prefix[6:0], opcode[7:0]} ==? 15'bXXXX1XX_00XX0001) pla[83]=1'b1; else pla[83]=1'b
  if ({prefix[6:0], opcode[7:0]} ==? 15'bXXXX1XX_00XX0001) pla[84]=1'b1; else pla[84]=1'b
  if ({prefix[6:0], opcode[7:0]} ==? 15'bXXXX1XX_00XX0001) pla[85]=1'b1; else pla[85]=1'b
  if ({prefix[6:0], opcode[7:0]} ==? 15'bXXXX1XX_00XX0001) pla[86]=1'b1; else pla[86]=1'b
  if ({prefix[6:0], opcode[7:0]} ==? 15'bXXXX1XX_00XX0001) pla[87]=1'b1; else pla[87]=1'b
  if ({prefix[6:0], opcode[7:0]} ==? 15'bXXXX1XX_00XX0001) pla[88]=1'b1; else pla[88]=1'b
  if ({prefix[6:0], opcode[7:0]} ==? 15'bXXXX1XX_00XX0001) pla[89]=1'b1; else pla[89]=1'b
  if ({prefix[6:0], opcode[7:0]} ==? 15'bXXXX1XX_00XX0001) pla[90]=1'b1; else pla[90]=1'b
  if ({prefix[6:0], opcode[7:0]} ==? 15'bXXXX1XX_00XX0001) pla[91]=1'b1; else pla[91]=1'b
  if ({prefix[6:0], opcode[7:0]} ==? 15'bXXXX1XX_00XX0001) pla[92]=1'b1; else pla[92]=1'b
  if ({prefix[6:0], opcode[7:0]} ==? 15'bXXXX1XX_00XX0001) pla[93]=1'b1; else pla[93]=1'b
  if ({prefix[6:0], opcode[7:0]} ==? 15'bXXXX1XX_00XX0001) pla[94]=1'b1; else pla[94]=1'b
  if ({prefix[6:0], opcode[7:0]} ==? 15'bXXXX1XX_00XX0001) pla[95]=1'b1; else pla[95]=1'b
  if ({prefix[6:0], opcode[7:0]} ==? 15'bXXXX1XX_00XX0001) pla[96]=1'b1; else pla[96]=1'b
  if ({prefix[6:0], opcode[7:0]} ==? 15'bXXXX1XX_00XX0001) pla[97]=1'b1; else pla[97]=1'b
  if ({prefix[6:0], opcode[7:0]} ==? 15'bXXXX1XX_00XX0001) pla[98]=1'b1; else pla[98]=1'b
  if ({prefix[6:0], opcode[7:0]} ==? 15'bXXXX1XX_00XX0001) pla[99]=1'b1; else pla[99]=1'b
  if ({prefix[6:0], opcode[7:0]} ==? 15'bXXXX1XX_00XX0001) pla[100]=1'b1; else pla[100]=1'b
  if ({prefix[6:0], opcode[7:0]} ==? 15'bXXXX1XX_00XX0001) pla[101]=1'b1; else pla[101]=1'b
  if ({prefix[6:0], opcode[7:0]} ==? 15'bXXXX1XX_00XX0001) pla[102]=1'b1; else pla[102]=1'b
  if ({prefix[6:0], opcode[7:0]} ==? 15'bXXXX1XX_00XX0001) pla[103]=1'b1; else pla[103]=1'b
  if ({prefix[6:0], opcode[7:0]} ==? 15'bXXXX1XX_00XX0001) pla[104]=1'b1; else pla[104]=1'b
end

```

The image shows the start of the PLA decode module implemented in Verilog.

Z80 has several opcode tables and addressing modes selected either by a combination of instruction prefix bytes (0xCB, 0xED and IX/IY) or by the internal state (HALT, ALU...)

PLA checker tool lets you set or unset any of these modifiers:



The modifier buttons directly correspond to modifiers in the PLA table and let you simulate the exact PLA logic behavior as you are executing various tool dumps.

The tool keeps a history of commands that are typed in; a number displayed at the front of a *prompt* “>>>” is a location in the history buffer. Pressing **PgUp** and **PgDown** selects a command from the history buffer; **ESC** clears the command line.

Arduino Tools

Directory “**tools\Arduino\Z80_dongle**” contains firmware for the *Arduino Mega* connected to a Zilog Z80 through a custom dongle. This setup can be used to pace Z80 in a controlled way and to execute individual instructions and monitor bus activity. You can read more about that dongle at www.baltazarstudios.com.

It was heavily used to generate tables for the correct bus behavior. These tables and Python scripts to create them are checked in the directory “**tools\dongle**”.

Integration

This section describes how to integrate A-Z80 CPU into your own project.

The method is tested with Altera and Xilinx design tools on several test boards. Other users contributed by using a variety of boards, including Lattice-based FPGA boards.

The process of integration involves adding all relevant source files and setting up interfaces. For convenience, all files needed to synthesize a CPU are listed in the file “cpu/top-level-files.txt” but a Python script “**cpu/export.py**” should be run to copy all design files to your project folder.

```
Usage: export.py <destination-folder>
```

```
Copies all core A-Z80 Verilog files to a destination folder of your choice.
```

This example shows how to run it to copy files to your project folder:

```
python.exe export.py c:\projects\board\cpu
```

```
Copying control/clk_delay.v
```

```
Copying control/decode_state.v
```

```
. . .
```

```
Copying toplevel/coremodules.vh
```

```
Copying toplevel/globals.vh
```

```
Done copying 49 files.
```

```
All necessary A-Z80 CPU files are copied to c:\projects\board\cpu
```

```
Add all Verilog files (*.v) to your project and ensure that Verilog include files (*.vh) are on the include path.
```

```
Use z80_top_direct_n.v as your top-level interface file.
```

```
Note for the users of Lattice FPGA toolset: instead of data_pins.v, manually copy and use data_pins_lattice.v file instead.
```

Several fully working sample implementations (*basic host* and *zxspectrum*) should provide good starting point.

Interface

The top-level file “z80_top_direct_n.v” exports the following interface:

```
module z80_top_direct_n(  
    output wire nM1,  
    output wire nMREQ,  
    output wire nIORQ,  
    output wire nRD,  
    output wire nWR,  
    output wire nRFSH,  
    output wire nHALT,  
    output wire nBUSACK,  
  
    input wire nWAIT,  
    input wire nINT,  
    input wire nNMI,  
    input wire nRESET,  
    input wire nBUSRQ,  
  
    input wire CLK,  
    output wire [15:0] A,  
    inout wire [7:0] D  
);
```

This interface pinout is 100% identical to the Zilog Z80 package pins. It implements Z80 bus timings and features tri state signals and buses. (While admittedly not optimal for an FPGA implementation, it perfectly mimics the actual Z80 silicon which was one of the goals of this project).

Example Implementations

Several working implementations are included. They are all located in the “**host**” directory and use Altera DE1 and Xilinx Nexys3 development boards.

Warning: The synthesis and *fMax* numbers as shown might vary depending on your tool version, applied timing constraints and the exact configuration.

Simple host

“*basic host*” project has two targets: “*basic_de1*” can be used on a Terasic DE1 board and “*basic_nexys3*” can be used on a Digilent Nexys 3 board. Each contains the A-Z80 CPU, 16 KB RAM configured as single port RAM and a unidirectional implementation of the UART for the text output. Each target project also includes a corresponding verification ModelSim configuration (for Altera) or an ISim test bench (for Xilinx).

File	Description
host\basic_de1\basic_de1.qpf	Quartus project file
host\...\simulation\modelsim\test_host.mpf	ModelSim project file for simulation
host\basic_de1\basic_de1_fpga.sv	Top-level board source file for FPGA implementation
host\basic_de1\basic_de1_ModelSim.sv	Top-level board source file for ModelSim board model
host\basic_de1\test_host.sv	ModelSim test bench for the simulation

File	Description
host\basic_nexys3\basic_nexys3.xise	Xilinx ISE project file
host\basic_nexys3\basic_nexys3_fpga.v	Top-level board source file for FPGA implementation
host\basic_nexys3\test_host.v	Test bench for the iSim simulation tool

This host board can load and run any Z80 executable (for example, one of those in “**tools\zmac**” directory). Programs can print to UART and, on a physical DE1 or Nexys3 boards, the text is seen through the attached serial terminal.

When run within a simulation environment, the text is written to a ModelSim or ISim output windows.

The following image shows the output of “**tools\zmac\hello_world.asm**” being captured through a serial port:

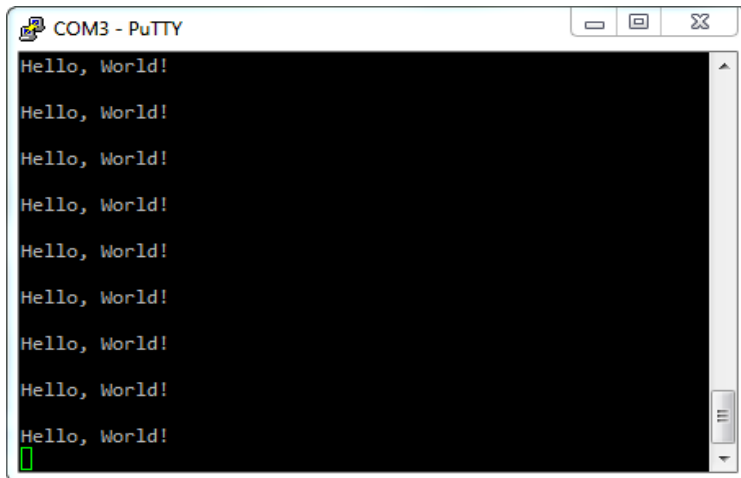


Image 2: "Hello, World"

Synthesis result of this simple design on an Altera DE1 board shows only 11% of the total LE resources used:

Flow Summary	
Flow Status	Successful - Sat Mar 12 14:37:51 2016
Quartus II 64-Bit Version	13.0.1 Build 232 06/12/2013 SP 1 SJ Web Edition
Revision Name	basic_de1
Top-level Entity Name	host
Family	Cyclone II
Device	EP2C20F484C7
Timing Models	Final
Total logic elements	2,084 / 18,752 (11 %)
Total combinational functions	2,019 / 18,752 (11 %)
Dedicated logic registers	400 / 18,752 (2 %)
Total registers	400
Total pins	11 / 315 (3 %)
Total virtual pins	0
Total memory bits	131,072 / 239,616 (55 %)
Embedded Multiplier 9-bit elements	0 / 52 (0 %)
Total PLLs	1 / 4 (25 %)

The CPU CLK is derived from the *pll_clk* and the effective Slow Model *fMax* for this compilation is 18.89 MHz.

Slow Model Fmax Summary				
	Fmax	Restricted Fmax	Clock Name	Note
1	18.89 MHz	18.89 MHz	clk_cpu	
2	50.66 MHz	50.66 MHz	CLOCK_50	
3	82.49 MHz	82.49 MHz	pll_[altpll_component]pll clk[0]	

Xilinx ISE tools synthesize the design using about 19% of Nexys3 device slice resources:

Selected Device: 6slx16csg324-2				
Slice Logic Utilization:				
Number of Slice Registers:	396	out of	18224	2%
Number of Slice LUTs:	1819	out of	9112	19%
Number used as Logic:	1819	out of	9112	19%
Slice Logic Distribution:				
Number of LUT Flip Flop pairs used:	2113			
Number with an unused Flip Flop:	1717	out of	2113	81%
Number with an unused LUT:	294	out of	2113	13%
Number of fully used LUT-FF pairs:	102	out of	2113	4%
Number of unique control sets:	70			
IO Utilization:				
Number of IOs:	37			
Number of bonded IOBs:	36	out of	232	15%
Specific Feature Utilization:				
Number of Block RAM/FIFO:	8	out of	32	25%
Number using Block RAM only:	8			
Number of BUFG/BUFGCTRLs:	3	out of	16	18%

Sinclair ZX Spectrum

This project implements a *Sinclair ZX Spectrum 48K* computer on an Altera DE1 board. See directory “**host\zxspectrum_de1**” containing a Quartus project file.

Directory “**host\zxspectrum\ula**” contains drivers for a PS/2 keyboard, video (using a VGA port), sound, RAM memory and clocks. Those were kinds of functions handled by Spectrum’s custom Ferranti ULA chip.

There are 2 system ROM images included in the “**host\zxspectrum\rom**” directory – the original ZX Spectrum ROM and an improved, so-called “Gosh Wonderful” ROM – merged into a single image which is to be flashed into the DE1’s flash memory starting at the address 0. Use a flash tool that came with your DE1 board software to flash this data.

The following table shows the function of buttons and switches. When a switch is activated, a **red LED** above it glows.

Button and Switch	Description
KEY0	Reset
KEY1	Issues NMI
SW0	Selects “Gosh Wonderful ROM” image versus the original ROM image
SW1	Disables interrupts
SW2	Turbo mode (3.5 MHz x 2 = 7.0 MHz)

Function of **green LEDs** is to show:

GREEN LED	Description
LEDG0-LEDG4	Kempston joystick UP, DOWN, LEFT, RIGHT, FIRE is pressed
LEDG5	A key is pressed
LEDG6	When blinking, a speaker or line-in is active

If you decide to try this design on your DE1 board and then want to play actual games, connect an Android device earphone jack out to DE1’s LINE-IN and use one of several apps to “play” a game into the device. For example, Baltazar Studio’s **PlayZX** was written for that purpose only:

<http://play.google.com/store/apps/details?id=com.baltazarstudios.playzxtapes>



Image 3 : Sinclair ZX Spectrum on Altera DE1

Image 3 shows a game “Manic Miner” being loaded through the audio line-in connector into the FPGA board visible in the middle and a Kempston compatible joystick in the foreground.

This is a synthesis result of a ZX Spectrum host design on an Altera DE1 board:

Flow Summary	
Flow Status	Successful - Sat Mar 12 15:01:43 2016
Quartus II 64-Bit Version	13.0.1 Build 232 06/12/2013 SP 1 SJ Web Edition
Revision Name	zxspectrum_de1
Top-level Entity Name	zxspectrum_board
Family	Cyclone II
Device	EP2C20F484C7
Timing Models	Final
Total logic elements	2,457 / 18,752 (13 %)
Total combinational functions	2,336 / 18,752 (12 %)
Dedicated logic registers	598 / 18,752 (3 %)
Total registers	598
Total pins	153 / 315 (49 %)
Total virtual pins	0
Total memory bits	131,072 / 239,616 (55 %)
Embedded Multiplier 9-bit elements	0 / 52 (0 %)
Total PLLs	1 / 4 (25 %)

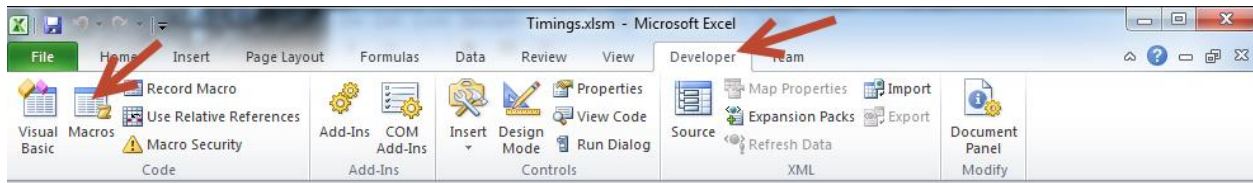
Advanced Topics

Modifying the A-Z80 CPU Core

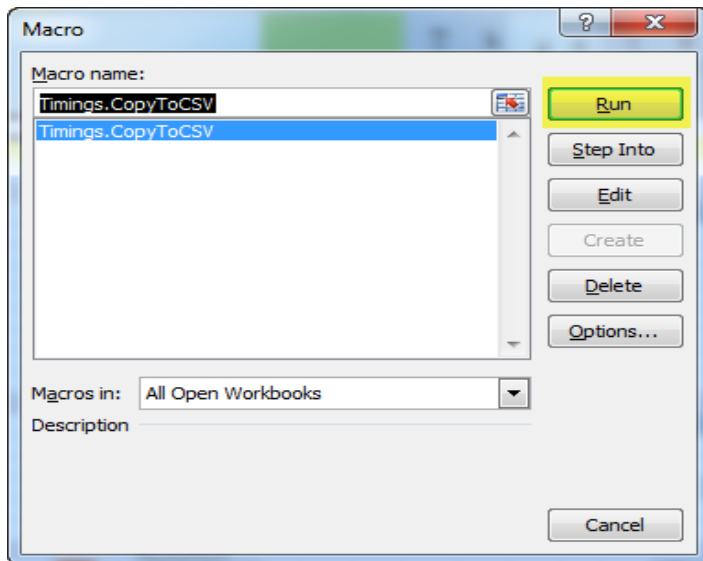
If you want to make a change to any instruction's timing or a sequence of micro-operations, modify "**cpu\control\Timings.xlsm**" file. This is a Microsoft Excel spreadsheet that contains timing tables for each group of instruction as decoded by opcode PLA and sequencer state. Vertical columns contain operations on specific CPU blocks. Instruction groups are listed by the **M** and **T**-clocks providing the exact timing for each set of operations.

Micro-operations are represented by short tokens (for example, "PC" or "mr") which are defined in the file "**cpu\control\timing_macros.i**". Every token is translated into one or more discrete control signals causing desired operation.

After changing the timing spreadsheet, export it into a TAB-delimited file. The spreadsheet contains a macro that will do it for you: click on the "Developer" menu and run Macros:



Running "CopyToCSV" macro will replace the existing CSV file with changed timings.



The next step is to create a Verilog file based on those timings by running a python script "**cpu\control\genmatrix.py**". That script reads in the CSV file containing timing tables and generates "**exec_matrix.vh**" that implements actual Verilog code to control the timings.

Although Altera Quartus synthesis tool can use that file to compile a working design, Xilinx synthesis tool is not able to handle its complexity, so for Xilinx devices an additional step is needed which takes that file and converts it to a format their tools are able to handle. Run **“cpu\control\gencompile.py”** to generate **“exec_matrix_compiled.vh”** file that defines the same set of combinatorial operations but in a sum-of-product terms.

All Python scripts in this project can be run in-place without the need to specify any arguments. You can run them either through a command line or by simply double-clicking on them.

If you change any *schematic file*, where your change adds or removes global input or output signals, you need to run three Python scripts to recreate various lists of global includes:

“cpu\control\gencoremodules.py” – generates instantiation file for all modules:

- “core.vh” contains all instantiated CPU modules

“cpu\control\genref.py” – generates global include files using all exported module signals:

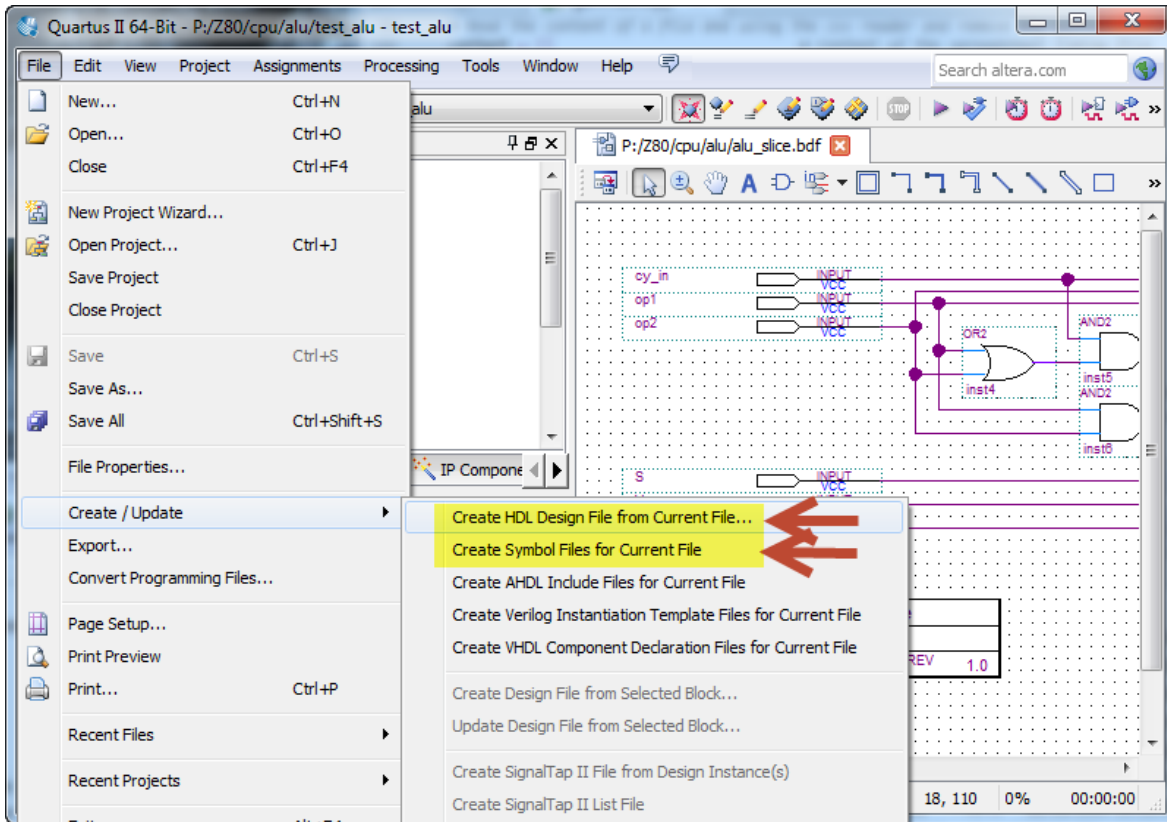
- “exec_module.vh” contains input/output definitions to be included in the module def.
- “exec_zero.vh” contains Verilog code to set all input wires to zero.

“cpu\toplevel\genglobals.py” – generates a list of global *wire* defines:

- “globals.vh” contains Verilog code that defines all global signal wires.

Quartus project files (*.qpf, *.qsf) in **“cpu\alu”**, **“cpu\bus”**, **“cpu\control”** and **“cpu\registers”** directories are non-functional and just conveniently hold sets of files together within a conceptually modular overall design. *Quartus* project in the **“cpu\toplevel”** directory only contains a top-level schematic diagram which is not functional. Hence, they are only containers to hold files.

When modifying a schematic file (and most of the A-Z80 blocks are designed at the schematic level), open a corresponding *Quartus* container project (for example, when modifying a schematic in the ALU block, open **“cpu\alu\test_alu.qpf”**). Change the schematics, compile it (to make sure it has no errors) and then export it to both the Verilog equivalent and a symbol file, as shown below:



Verilog versions are used as final target files that compile with the rest of the A-Z80 core files while symbol files are (at the moment) optional but could be used in the future to create a fully working schematic top-level (TBD).

File Generators

This section describes Python scripts and processes that generate various files. You should never edit or change generated files since they will be overwritten the next time scripts are run. Many scripts have additional options, but the defaults are meaningful, so the scripts can be run in very simple ways (for example, by simply double-clicking on them).

See each individual script for more information.

Building the CPU

Process:	cpu/control/Timings.xlsm
Generates file:	cpu/control/Timings.csv
Uses files:	-

This is a *MS Excel* spreadsheet that defines a matrix of timings vs. operations for each group of instructions. The spreadsheet contains an embedded macro that exports that data in a suitable format; go to the Developer tab, and in the Code group, click on Macros. Run a macro "Timings.CopyToCSV".

Process:	cpu/control/genmatrix.py
Generates file:	cpu/control/exec_matrix.vh
Uses files:	Timings.csv, timing_macros.i

This script reads the A-Z80 instruction timing data from a (generated) spreadsheet text file and creates a Verilog include file which defines the control block execution logic matrix. Token keywords in the timing spreadsheet are substituted using a list of keys stored in the timing macros file.

Process:	cpu/control/gencompile.py
Generates file:	cpu/control/exec_matrix_compiles.vh, temp_wires.vh
Uses files:	cpu/control/exec_matrix.vh

This script reads the exec matrix file and compiles it into an alternate format that can be used with Xilinx tools.

Process:	cpu/control/genref.py
Generates file:	cpu/control/exec_module.vh, exec_zero.vh
Uses files:	top_level_files.txt + list of files in that file

This script reads a list of files from the “top_level_files.txt” and processes each file to extract selected names of wires defined in their respective modules to be used to easily define and initialize them.

Process:	cpu/toplevel/gencoremodules.py
Generates file:	cpu/toplevel/coremodules.vh
Uses files:	top_level_files.txt + list of files in that file

This script reads and parses all top-level modules and generates a core block file containing instantiation of these modules in Verilog format. The generated file is included by core.vh.

Process:	cpu/toplevel/genglobals.py
Generates file:	cpu/toplevel/globals.vh
Uses files:	top_level_files.txt + list of files in that file

This script reads a list of files from the “top_level_files.txt” and processes each file to extract selected names of wires defined in their respective modules to be used as *global* wires.

Process:	tools/z80_pla_checker.exe
Generates file:	Verilog PLA decode module
Uses files:	PLA and opcode defines from resource folder

This executable program reads a PLA table in a simple format (reverse-engineered from a picture of a die) and generates a Verilog PLA decode module.

Verification/Tests

Process:	cpu/toplevel/genfuse.py
Generates file:	cpu/toplevel/test_fuse.vh
Uses files:	fuse/tests.in, tests.expected, regress.in, regress.expected

This script creates a *Fuse* test in Verilog from several test description files.

Process:	tools/zmac/make_fpga.bat, make_modelsim.bat
Generates file:	Compiled HEX files
Uses files:	Z80 assembler source files

These DOS batch files compile and create HEX files from any Z80 assembler source files dropped onto it. They speed up running and testing of Z80 programs.